



SPARK and Abstract Interpretation – A White Paper

Rod Chapman

SPARK and Abstract Interpretation—A White Paper

Dr. Roderick Chapman
SPARK Products Manager
Praxis Critical Systems

September 2001

Introduction

Recently, there has been significant interest in the use of Abstract Interpretation (AI) technology in the static analysis of critical software. A number of AI-based tools exist, but some of their marketing suffers from a level of hyperbole that is at best optimistic, and at worst somewhat irresponsible.

There have also been some attempts to compare AI-based static analysis tools with the analysis implemented by the SPARK¹ language and SPARK Examiner toolset. The aim of this white paper is to dispel some of the common myths and to avoid potential confusion with customers.

Static Analysis—The need for precision and early use

Static analysis of software is a well-known and widely-used technique. Static analysis ranges from simple manual inspections through to full proof of program behaviour. While manual forms of analysis are still useful, a real qualitative change in analytical power occurs when analysis is both *automated* and the notation being analyzed has *precise* semantics.

Precision is all about lack of ambiguity. All modern standard programming languages (even Ada) suffer from ambiguity in their definition. For instance, in Ada the order of evaluation of expressions and the parameter passing mechanism are not defined, to allow the language to be reasonably implemented by a range of compilers for a range of target architectures. This is perfectly reasonable, but creates a significant problem for a static analysis tool. In the face of such ambiguity, a tool must either analyse *every* possible option, or must make an *assumption* in choosing one of them.

Both of these options suffer a severe penalty. If a tool attempts to analyse every possible outcome of every ambiguity, then this typically leads to an explosion in analysis time and space that renders the analysis useless. In the second option, the results of the analysis are only valid if the user's compiler happens to make the same choice as the analysis tool.

SPARK suffers from neither of these problems, since SPARK (as a very carefully defined and annotated subset of Ada) eliminates all ambiguity from the language. This enables analysis to be efficient, and the results are valid for all compilers.

A second well-known benefit of static analysis is that it enables errors to be detected and eliminated *early* in the software lifecycle. This is certainly true, and the economic benefits of such analysis are widely recognized, particularly in the construction of high-integrity and critical systems where the cost of late error detection is dramatic. To this end, static analysis should be applied *early* and *often* in the development of designs and code. By *early* we really mean “before the code is ever compiled” and this is possible in SPARK owing to the design of the language and the Examiner, which allows effective analysis of incomplete programs at very high speed.

¹ Note: The SPARK programming language is not sponsored by or affiliated with SPARC International Inc. and is not based on the SPARC™ architecture.

AI—Some issues to consider

AI-based tools are able to implement a number of types of static analysis. In particular, these tools have concentrated on analysis of “runtime errors”, the most important class of which are the predefined exceptions of Ada. In evaluating these tools, the following points should be considered:

“Don’t Know” answers

In attempting to answer the question “Is this statement free from predefined exceptions?” an AI-based analysis for a full language such as Ada can (and indeed in certain circumstances *must*) answer “Don’t Know.” The number of such answers clearly affects the usefulness of the results, and depends crucially on the quality of the software and the language subset under analysis. One AI tool vendor has reported excellent results but, somewhat ironically, for software that is already written in SPARK.

Ambiguity and assumptions

As pointed out above, the inherent ambiguity in the definition of full Ada limits any attempt at static analysis of the full language. Users must be aware of this in evaluating such tool support. Where a tool makes assumptions, such as evaluation order and parameter passing mechanism, these must be carefully considered to determine their validity. Examples of such ambiguous programs, and the problems associated with them, are given in Appendix A.

Numerical algorithms

We know from our own experience that proving exception freedom for real-number (i.e. floating and fixed point) algorithms is extremely difficult. Floating point arithmetic is a rich source of ambiguity in programming language design, and so the precise semantics of such algorithms are dependent on compiler- and target-dependent features.

While the SPARK Examiner and proof tools are capable of reasoning about *rational* (not floating point) values, this facility is not enabled in the SPARK Examiner by default, since it has to be used with such care.

In light of these points, an AI-based tool cannot claim to detect “100%” of runtime errors.

An historical note

While someone had to be first to market, there are at least two tools we know of that implement this style of AI-based search for exception freedom, so the technology is certainly not unique. SPARK’s approach to proof of exception freedom was first published in 1993, has been implemented in the SPARK toolset since July 1996, and continues to be widely used.

AI and SPARK

Comparisons between AI-based tools and the analyses implemented by the SPARK Examiner must be considered in the light of the observations above. A tool which attempts to analyse a whole language like Ada differs in design hugely from the design of the Examiner, which operates on a smaller, but nevertheless very rich, unambiguous language. The Examiner does indeed implement a form of analysis that allows SPARK programs to be proven (in the mathematical sense) to be free from Ada’s predefined exceptions, but this is only one of its many functions.

One other myth—that you need a PhD in Maths to do proof of software—does warrant some comment here. Most proof activity these days is done by automated theorem proving tools (for example, the SPARK Simplifier tool). What you need to do proof is CPU cycles—lots of ‘em—but these are cheap—*far* cheaper than the time and energy of engineers. For well-written SPARK, the Simplifier typically discharges some 90–95% of the verification conditions for proving exception freedom. Some skill is needed to interpret and review the remaining VCs, but this is readily obtainable by any graduate level engineer. Many of our clients have found proof of exception freedom to be an effective activity and continue to use it, with or without our direct help.

Conclusions

This brief white paper has hopefully helped to dispel some of the myths and confusion surrounding AI-based static analysis and its comparison with SPARK.

It should also be clear that AI is a complementary technology, which certainly has a place in many environments. As with any tool, however, it is important to use it in the full knowledge of its limitations and assumptions.

If you would be interested to understand more about how these complementary technologies can be used, please contact us at:

Roderick Chapman MEng DPhil MBCS CEng
Products Manager
Praxis Critical Systems

sparkinfo@praxis-cs.co.uk
www.sparkada.com
www.praxis-cs.co.uk

Appendix - some example ambiguous programs

This appendix present some examples of ambiguity in programs that present particular difficulties to abstract-interpretation based static analysis.

Example 1 - Function side-effect and evaluation order dependency

Consider the following program:

```

with Text_IO; use Text_IO;
procedure Test1
is
  X, Y, Z, R : Integer;

  function F (X : Integer) return Integer
  is
  begin
    Z := 0;
    return X + 1;
  end F;

begin
  X := 10;
  Y := 20;
  Z := 10;

  R := Y / Z + F (X); -- order dependency here.

  Put (Integer'Image(R)); -- R = 13 if L->R evaluation,
                          -- constraint error if R->L
end Test1;

```

This program exhibits a function side-effect, since F updates the value of Z which is global to it. The assignment to R exhibits an evaluation order dependency (which also makes the program *erroneous* in Ada terminology) since if F(X) is evaluated first, the division Y / Z is required to raise Constraint_Error.

This program is, of course, illegal in SPARK, since a function in SPARK may not have a side-effect.

It is interesting to note that when compiled with GNAT 3.13p on Windows NT, this program raises Constraint_Error every time, whereas AI, depending on its implementation, may not uncover this error.

Example 2 - Aliasing of parameters

The following program illustrates another interesting case of ambiguity and erroneousness in Ada: that of aliasing. Consider:

```

with Text_IO; use Text_IO;
procedure Test2
is
  type Rec is record
    F, G : Integer;
  end Record;

  R      : Rec;
  Result : Integer;

  procedure Multiply (X, Y : in Rec;
                     Z   : out Rec)
  is
  begin
    Z := (0, 0);
    Z.F := X.F * Y.F;
    Z.G := X.G * Y.G;
  end Multiply;

begin
  R := (10, 10);
  Result := 100;
  Multiply (R, R, R); -- parameters overlap here.
  Result := Result / R.F;

  Put (Integer'image(Result)); -- Result = 1 if pass by copy
                                -- constraint error if pass by
                                -- reference

end Test2;

```

Here, the call to `Multiply(R, R, R)` results in *aliasing* of the formal parameters in the called subprogram body. If the record `Rec` is passed by copy (a perfectly valid choice for a compiler to make), then all is well, and the program prints “1”.

This program is illegal in SPARK, since “in out” parameters may not be overlapped in a procedure call in this way in SPARK. This problem is trivially detected by the Examiner.

Now consider the following slight extension to the program:

```
with Text_IO; use Text_IO;
procedure Test2
is
  type Rec is record
    F, G, H : Integer;
  end Record;

  R      : Rec;
  Result : Integer;

  procedure Multiply (X, Y : in Rec;
                    Z      : out Rec)
  is
  begin
    Z := (0, 0, 0);
    Z.F := X.F * Y.F;
    Z.G := X.G * Y.G;
  end Multiply;

begin
  R := (10, 10, 10);
  Result := 100;
  Multiply (R, R, R);
  Result := Result / R.F; -- Result = 1 if pass by copy
                        -- constraint error if R was passed by
                        -- reference
  Put (Integer'image(Result));
end Test2;
```

Here, a well-meaning engineer (perhaps months after the original code was written) has added a third field “H” to the record “Rec.” This means the compiler may choose to pass Rec by *reference* rather than by copy. In this case, the assignment “Z := (0, 0, 0);” will overwrite the formal parameters X and Y, so R.F is zero in the subsequent division.

When compiled with GNAT 3.13p on NT, this program raises Constraint_Error every time, but again this would be difficult for AI technology to determine.